A Universal Windows Bootkit: An Analysis of the MBR Bootkit "HDRoot"

William Showalter
Mississippi State University
was249@msstate.edu

Abstract

In October, 2015 Kaspersky released an analysis of the bootkit "HDRoot". Their analysis highlighted mistakes in the bootkit, which made it ineffective at performing its task. Upon attempts to replicate that analysis however, it appears that these conclusions were in error and the bootkit works with any Windows version in the last 16 years. HDRoot represents a serious commitment in time and effort to develop, and an in-depth analysis reveals the work of a significantly capable threat actor. The sample analyzed here dates to 2013, and is the same sample Kasperky reports to have analyzed in their post. However, all evidence points to Kaspersky performing analysis with a 2006 sample, likely the reason for their conclusions. Additionally, mistakes made in reporting the capability of offensive software, provided without means to verify, hurt the security industry by misleading practitioners and limiting their ability for informed decision making.

1. Introduction

Kaspersky attributes the HDRoot malware to the WINNTI group, a threat actor that they connect to Chinese origins [1]. This analysis looks at a dropper for HDRoot and the installed bootkit code, and does not attempt attribution or analyze any other samples linked to HDRoot. The dropper is capable of installing any PE executable as the payload for the bootkit, but does not come bundled with any default payloads. As such, this report offers no insight into the various payloads used by the authors in practice. For information on the other malware associated with the WINNTI group, see Trend Micro or Kaspersky's reports on the group [2] [1]. This analysis does, however, offer a very in-depth look into the technical workings of the HDRoot bootkit and its components.

Also addressed in this work are a number of technical inaccuracies and misrepresentations in Kaspersky's SecureList post of October 2015, "I am HDRoot! Part 1," which was the only research on this sample to be published prior to this paper [4]. Kaspersky's research was very helpful getting started, but it soon became apparent that their analysis was not actually performed using the sample identified by MD5 in the article and thus could not be relied upon. This is believed to be the reason for many of their criticisms of HDRoot, which they call, "not what you expect from such a serious APT actor." The sample analyzed here is not free of criticisms, but all of the problems addressed by Kaspersky appear to have been fixed in the sample their post lists the signature of. Other faults and weaknesses, however, are addressed.

1.1. Sample

MD5 Hash:
`2c85404fe7d1891fd41fcee4c92ad305`
SHA1 Hash:
`4c3171b48d600e6337f1495142c43172d3b01770`

Original Name: net.exe
Product Version: 6.1.7600.16385
Time Stamp: 2012/08/06 13:12:39 UTC
Produce Name: Microsoft Windows Operating System
Retrieved from malwr [3].

2. Overview of Architecture

The malware examined here can be broken into several stages. The 64-bit dropper, which was signed with a stolen certificate that has since been revoked, is the first component that is executed. The dropper installs the bootkit to the hard drive along with a backdoor executable to be run on subsequent boots. The backdoor is supplied as a command line parameter to the dropper and can be any Win32 or Win64 executable.

Upon boot, the computer will execute the maliciously installed MBR, which loads a subsequent component that is named here as the "verifier". It is a single sector block that verifies that the rest of the bootkit and the backdoor are intact before running them. The bulk of the bootkit's work is done by the next component, rkImage. The name rkImage actually comes from the interface of the dropper, which explicitly refers to it when installing the bootkit. rkImage works by manually reading the file system from the disk in order

URI: http://hdl.handle.net/10125/41896
ISBN: 978-0-9981331-0-2
CC-BY-NC-ND

to write the backdoor (the generic term used to refer to the payload) into the filesystem and redirect a Windows system service to launch the backdoor.

When rkImage is finished it transfers execution back to the original, non-infected MBR and allows Windows to boot normally. The booting system will run the backdoor instead of the replaced system service, but will then restore and start the legitimate service after the backdoor has ran, hiding the fact it was ever replaced.

3. Dropper

The dropper is designed to disguise itself as the Windows system utility net.exe. The properties on the executable attempt to mirror the settings found on a Windows 7 version of the utility, reporting it to be a Microsoft program. When run without parameters, HDRoot shows the options menu as if it were net.exe. That is where the similarities end, however.

The dropper executable, while masquerading as the Microsoft net command, has been signed with a digital certificate belonging to Guangzhou YuanLuo Technology Co, Ltd, a firm based in the city of Guangzhou, China who had their signing certificate stolen by the WINNTI group. The certificate has since been revoked, and, if the signing time and compilation dates on the executable are to be believed, it was signed in 2013 almost a year after this version was initially compiled.

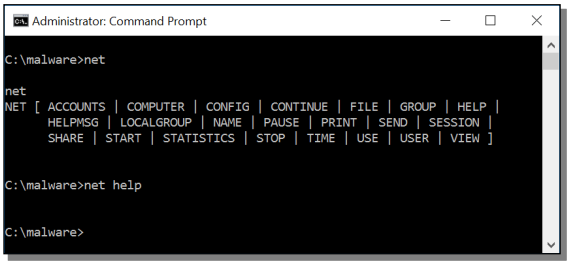

Figure 1. Dropper disguising itself as net.exe

3.1. VMProtect

A notable hindrance to reversing the dropper is that it was packed using VMProtect. Unlike most packers, which decompress and then jump to the original executable code, VMProtect converts the x86 opcodes into an automatically generated language of bytecodes to be interpreted in its own emulator. Attempting to statically analyze the sample would prove an arduous task. Instead of x86 instructions, the main contents of the program exist in the language of the automatically generated emulator, and the only intelligible instructions left are for interpreting these opcodes. There have been a few unpacking plugins for Ollydbg written for certain versions of VMProtect, but these are generally found in forum posts and are not well maintained. This is most likely the reason Kaspersky did the bulk of their analysis with a different sample that was almost, but not quite, functionally the same. Not wanting to spend time tackling VMProtect, a number of other techniques were used to analyze artifacts instead.

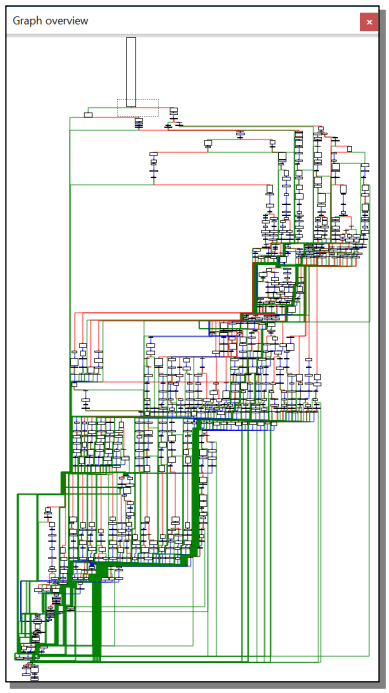

Figure 2. Graph overview of VMProtect's emulator.

When the dropper obscured with VMProtect is run with any of the legitimate net command parameters or with unrecognized parameters, no output is given. The only commands that provide output are the valid HDRoot commands programmed by the authors. Kaspersky, by analyzing an older sample from 2006, was able to get a “help” output, rather than the “net” output, which contained a list of commands for that version. This was not present in the 2013 sample analyzed here, unfortunately, and the VMProtect obfuscation makes finding a command list through normal reverse engineering means prohibitive. Most of the previous commands still worked on the newer sample, and all of the commands were five or less characters in length. Even short words like install were abbreviated to “inst”, rather than left intact.

Since the Kaspersky command listing was half a decade older than this sample, and some of the commands from their listing were no longer present, a simple, and very slow, fuzzer was written to attempt to check all possible commands of five or less characters.

Given the length of the other commands and that input appears to be case insensitive, this appears to be a reasonable approach. The code for the fuzzer can be found in the supplemental files on the author's website. Screenshots for each command can be found there as well. No additional commands to the ones Kaspersky detailed were found by the fuzzer and Table 1 contains the complete list of known commands.

Table 1. HDRoot dropper commands

Command	Description
check	Checks for the presence of the bootkit and the integrity if present.
clean	Removes the bootkit.
inst <Backdoor>	Installs the bootkit
info <Backdoor>	Shows information about the checksums and requirements for an executable if it was installed as the backdoor.

3.2. DEBUGFILE.sys – A signed kernel driver

At the time the dropper installs the bootkit, no changes to the filesystem or the registry are seen between snapshots taken before and after. The approach used was to run the dropper in a continuous loop in a virtual machine, suspending the VM, and analyzing the resulting memory image. Performing memory capture from outside the VM appeared to be the best option because there were a number of anti-debugging techniques employed along with the anti-disassembly. Using Volatility, two more PE files were discovered that the dropper extracted inside the process, but none of the four clear text resources Kaspersky claimed to have extracted from a memory dump, providing further proof that they did their analysis on a different sample than is listed in their blog post. The two PE files found were kernel drivers, one 32-bit and one 64-bit. The 64-bit driver is signed, as is required by 64-bit versions of Windows, using yet another stolen certificate, while the 32-bit driver is not signed. This certificate belongs to a South Korean video game company, Neowiz. The certificate, unlike the one for the dropper, has yet to be revoked.

The use of the kernel drivers is fairly straightforward. Without kernel access there is no way for malware to write directly to the physical disk as there are no Windows API calls available to user land processes for doing so. The dropper writes out the appropriate driver to the file DEBUGFILE.sys in C:\Windows\system32\Drivers, and then creates a service for it. This shows up in the memory image as a registry handle to HKEY Local Machine with the path System\ControlSet001\services\DEBUGFILE. The service runs and the driver \Driver\DEBUGFILE is created. DEBUGFILE.sys is also deleted from the disk. The driver is used by the dropper to proxy its direct access to the physical disk. A number of things are done in this process. The original MBR is backed up and then overwritten by the new bootkit MBR, and then weakly encrypted components are written to disk. Near the beginning of the disk is the component I've named the verifier, followed by two identical copies of the original MBR. In another section of the disk is the main component of the bootkit, rkImage, followed by the backdoor that was installed.

One peculiar thing the malware does is install a second copy of the rkImage and backdoor files. This copy is encrypted identically to the first, and positioned such that it ends exactly 2063 sectors from the end of the drive. What makes this strange is that nothing in the bootkit will ever transfer execution to the second copy, and that the second copy is only installed if the drive has at least 30% free space. Kaspersky erroneously identified this behavior as only installing if the disk has greater than 30% free space, rather than installing a redundant copy of itself. As can be seen in a Windows 7 screenshot from the supplemental files, the bootkit is perfectly capable of installing with less than 30% free space. The only guess the authors make as to the purpose of this second copy is for the indented backdoor to be able to identify if one of the copies has been modified after it starts. The dropper will also detect a modified copy with the "check" command.

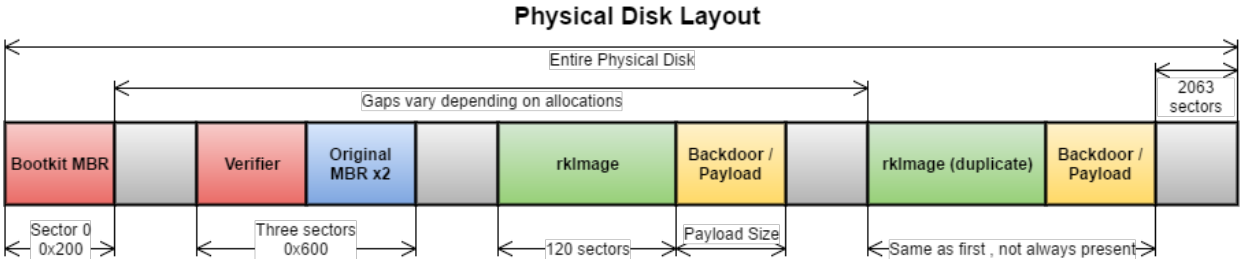

Figure 3. Physical disk layout written by DEBUGFILE.sys

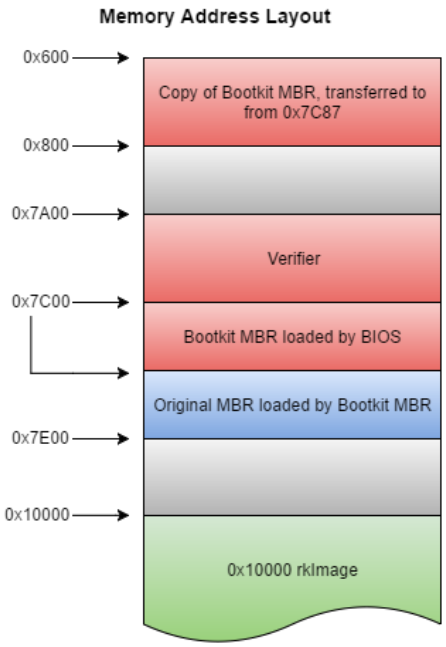

Figure 4. Address layout of memory loaded before rkImage

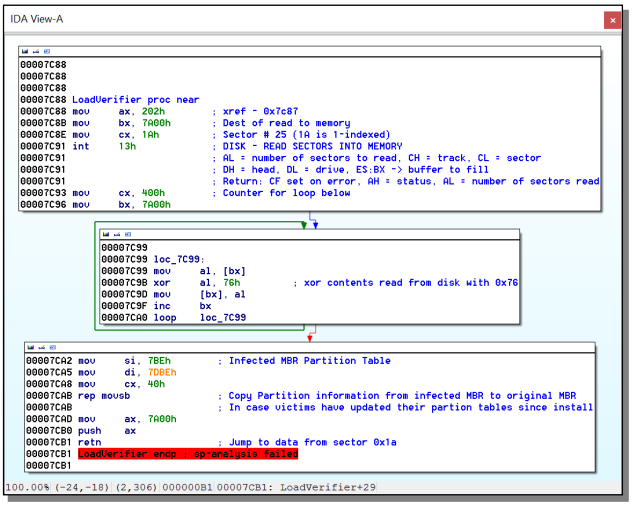

Figure 5. Infected MBR code to load, decrypt verifier

4. MBR

For all x86 systems not running UEFI, the boot process starts with the BIOS loading the Master Boot Record into memory and jumping to it. By convention, the BIOS loads the MBR to the physical memory address of 0x7C00. Another convention that many MBRs follow is to copy themselves, a single 0x200 sized sector, to the address 0x600 and then transfer execution to this location. The HDRoot bootkit is no exception. This is partly because the only code it actually changes in the original MBR is the jump address and the code jumped to. Most of the original MBR and partition table information is intact.

A normal MBR would look at the partition table to find the partition with the boot flag set, and then load the volume boot sector of that partition and transfer execution. HDRoot's MBR works similarly by calling interrupt 13 to read two sectors from disk into memory at the address 0x7A00 (through 0x7DFF). These are the verifier and the original MBR, which now has been loaded into the location where the MBR would have originally loaded on a non-infected system. The bootkit does not store these on disk in clear text, however. They are written to disk having been XOR'd with the byte value 0x76. The supplemental files have a C utility that can be used to decrypt the values. A function at offset MBR+0x88 performs these read and decrypt operations, copies the partition table from the infected MBR to the original (incase the victim has changed any partitions since the bootkit was installed), and then transfers execution to the verifier.

5. Verifier

The job of the verifier is to make sure that the bootkit is intact and that a specific set of criteria are met before allowing the bootkit to run. If any of these criteria fail the verifier will transfer the boot process to the original MBR, now at 0x7C00, without the bootkit executing. This mechanism helps prevent bricking the victim machine in the event that one or more of the bootkit sectors are corrupted or overwritten.

The first criteria in the verifier process is a check for whether the alt key is pressed on the keyboard. If the alt key is pressed, the bootkit launch will be aborted. The verifier then checks for a value at 0x7A08 (+0x8 from the verifier start address). If the value is null, the startup is aborted. This value is the drive identifier of where rkImage and the backdoor are stored. This was 0x80 in all the systems tested, which indicates drive 0. The dropper sets this value, and the subsequent bytes, before writing them to disk. This ensures that the bootkit was properly setup during install, and allows for the bootkit to be stored on a separate disk from the system disk. Table 2 shows a breakdown of the rkImage information stored in the verifier.

The third and final check done by the verifier is computing a CRC16 value on the obfuscated contents of rkImage and the backdoor (still only encrypted with an XOR 0x76). It compares the results to the saved CRC, and if they do not match it aborts. Otherwise it reads the entire rkImage, but not the backdoor, to 0x10000, and

then decrypts both. The last step is to copy the size and location information to the start of rkImage, so it can locate the backdoor for installing.

Table 2. rkImage location information in verifier

Address	Contents
0x7A02	0x55AA, not used, signals start of rkImage location data.
0x7A04	CRC16 value for rkImage+Backdoor.
0x7A06	Sector count for rkImage+Backdoor.
0x7A08	Drive number
0x7A09	Sector where rkImage+Backdoor starts.
0x7A0D	Next 0x55AA value, if present
…	…

6. rkImage

A significant component to reverse engineering the functionality of this bootkit was becoming familiar with the mechanics of low-level, pre-OS x86. For anyone looking to get into this the Intermediate Intel x86 videos on OpenSecurityTraining.info are highly recommended [5]. Even just following the transition from the verifier to rkImage requires some understanding of these mechanics, as the processor is still in 16-bit real mode at this time and a far jump is being performed, crossing a barrier between segments. This seems like a trivial thing until you find out that GDB, even operating in 8086 mode remotely debugging the bootkit running in QEMU, has absolutely zero understanding of segment addressing and completely falls apart trying to set breakpoints at any address higher than 0xFFFF. In retrospect Bochs might have been a better choice for this over QEMU, but not being familiar with it either the author struggled through with QEMU, performing most of the analysis indirectly, either statically or dynamically through the clues left behind by the bootkit's actions.

The first task rkImage sets itself to, like any sane bootloader, is to transfer itself from real mode to protected mode and then to 32-bit mode. In order to enable protected mode, the Global Descriptor Table must be setup and loaded. This is actually fairly unimportant to the operation of the malware but understanding it helped with getting the disassembly properly setup in IDA Pro to assist with the process. The details regarding and the contents of the segment descriptors are outside of the scope of this paper, but for anyone looking to make an analysis of them, the clearest explanations and diagrams were found in the AMD Architecture Programmer's Manual, Vol 2., for system programming [6]. Something that caused confusion was that most diagrams detailing the structure of segment descriptors (the entries in the Global Descriptor Table) are for the descriptors in 64-bit mode, since the 32-bit descriptors, called legacy segment descriptors in AMD's documentation, have a different structure. The work done reassembling the GDT can be seen in the rkImage IDB file in the supplemental files.

Once setup in a 32-bit environment, rkImage decrypts two sections in itself, the first containing a 32-bit DLL, and the second containing a 64-bit DLL. These are used to launch the backdoor from a Windows system service upon Windows booting. Each DLL has a 4-byte XOR key. The data is stored in rkImage in the format: 4-byte key, 4-byte length, encrypted PE file contents. The 32-bit DLL and its data are located at an offset of 0x4BE0 and the 64-bit values are at 0x6DE8, immediately after the previous DLL. These DLLs as packaged contain the registry path to the LanmanServer service DLL. The version of rkImage in this sample, installed by the 64-bit dropper, overwrites the LanmanServer information with the paths and values for the Schedule service. This allows for changing the target service without the need to recompile the DLLs embedded in rkImage. Kaspersky's observations that there are a number of different services that different samples have targeted supports this conclusion.

The backdoor executable is then loaded into memory and decrypted with the 0x76 XOR operation. At the end of this preparation work of loading, decrypting, and copying data, rkImage calls a function that is marked as DETERMINE_VERSION_NT_32_64_BIT in the supplemental disassembly. This is the function that determines what Windows version is installed, what malicious DLL to use, and where to install it. Since the malware will attempt to boot in any Windows version from Windows 2000 to Windows 10, there is a considerable nest of switch and if statements happening here. It first checks whether there is a "\winnt" directory, which is present in Windows 2000, and then if "\winnt" is not found it will subsequently check in "\windows\system32\kernel32.dll". The check for kernel32.dll in system32 prevents the bootkit from continuing install on Windows 98 and lower systems, as kernel32.dll was stored in the system directory rather than system32 before this point. If kernel32.dll is found it will check for "\users" and "\documents and settings" to determine if it is XP/2003, or Vista or newer. If it is not able to locate any of these, it falls out of the switch statement and no bootkit is installed.

If the directory selected was not "\winnt", it will check for "\windows\syswow64" and set a variable indicating if the system is 64-bit. This is used later when choosing which DLL to install (which means it is even 64-bit Windows XP / Server 2003 compatible). Then for each of the three operating system categories it will write the backdoor to %TEMP%\Explorer.exe

Table 3. Service DLL Paths by OS, in order attempted

Windows Version	Path
Windows 2000	\winnt\help\access.hlp
Windows 2000	\winnt\system\OLESVR.DLL
Windows XP or 2003	\windows\twain.dll
Windows XP or 2003	\windows\system\OLESVR.DLL
Windows Vista/2008 & Newer	\windows\syswow64\C_932.NLS
Windows Vista/2008 & Newer	\windows\system\OLESVR.DLL
Windows Vista/2008 & Newer	\windows\syswow64\kmddsp.tsp
Windows Vista/2008 & Newer	\windows\syswow64\Irclass.dll

(wherever %TEMP% is located for that version of Windows), and iterate through a list of files. If the file is present it will copy the appropriate DLL into the beginning of the file, overwriting the contents already

there. The files to be overwritten in question are shown in Table 3, and appear to be carefully selected to cause the least potential problems, with all but one of them being for different architectures than the running host.

Windows 2000 will attempt to first overwrite the access.hlp file, which, if anyone has not already disabled the help popups, may cause errors. Similarly, the 16-bit OLESVR.DLL file is overwritten if access.hlp does not exist. This will only cause issues if it is used by a 16-bit

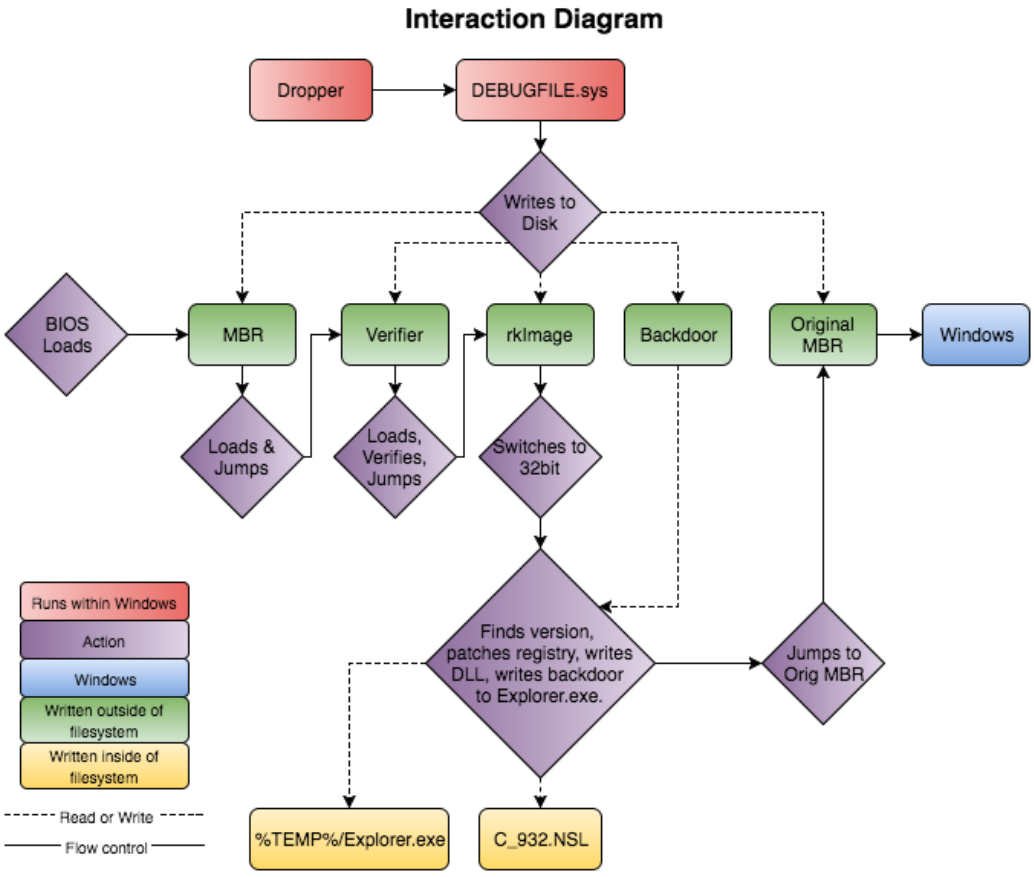

Figure 6. Diagram showing the out-of-OS boot process

application, as 32-bit applications will be using the system32\OLESVR32.DLL file. Windows XP will attempt to overwrite the 16-bit version of the twain.dll library for scanners (even using old scanners, twain32.dll should be used), and then the 16-bit OLESVR.DLL if the twain.dll is not found.

In 64-bit Windows Vista and newer systems, the default target is C_932.NLS, which is a 32-bit National Language Support file for the Japanese language [7]. This assumes that authors did not plan on infecting targets running 32-bit applications in Japanese, as this would cause issues. The only file that will be tried for 32-bit Windows Vista and newer is the same 16-bit OLESVR.DLL. This will only cause issues for applications run in 16-bit compatibility/emulation mode, as they are not natively supported in Vista and newer, and is therefore unlikely to affect most targets. The other two potential target files, which are also unlikely to be used, are both 16-bit DLLs found in the syswow64 (32-bit compatibility) directory. They are actually only labeled as compatible with Windows Server 2003 and earlier operating systems on MSDN, but are for some reason still included in the syswow64 directory. Kmddsp.tsp is a "kernel mode device driver" for "telephony service provider" network drivers, and IRClass.dll is an Infrared Class Coinstaller [8]. Neither should ever be used on a 64-bit system and therefore won't cause any issues if overwritten.

Once the DLL has been written to the appropriate file, the registry is patched to overwrite the Schedule service's DLL path with the path to the overwritten file. This should be approximately: HKLM\SYSTEM\CurrentControlSet\Services\Schedule\Parameters\ServiceDLL.

rkImage will then return to 16-bit real mode, handing execution back to the original MBR at 0x7C00, allowing the boot process to continue and Windows to load. It is worth noting that since Windows NT also used the C:\WINNT directory, it will match the first section of the bootkit which chooses the files to write the DLL into. However, since Windows did not introduce the svchost.exe process until Windows 2000, services did not have a Parameters sub-key or a ServiceDLL value in Windows NT. As such, if installed on Windows NT the bootkit wouldn't be able to locate the registry key for editing, and would fail out of the installation process. Additionally, it is likely that 32-bit versions of the dropper would not allow the install to a Windows 2000 system.

7. Schedule Service DLL

The final component of the bootkit, responsible for running the backdoor within Windows, is the DLL that replaced the Schedule service. The bootkit did not change the rest of the registry key, so it will be loaded

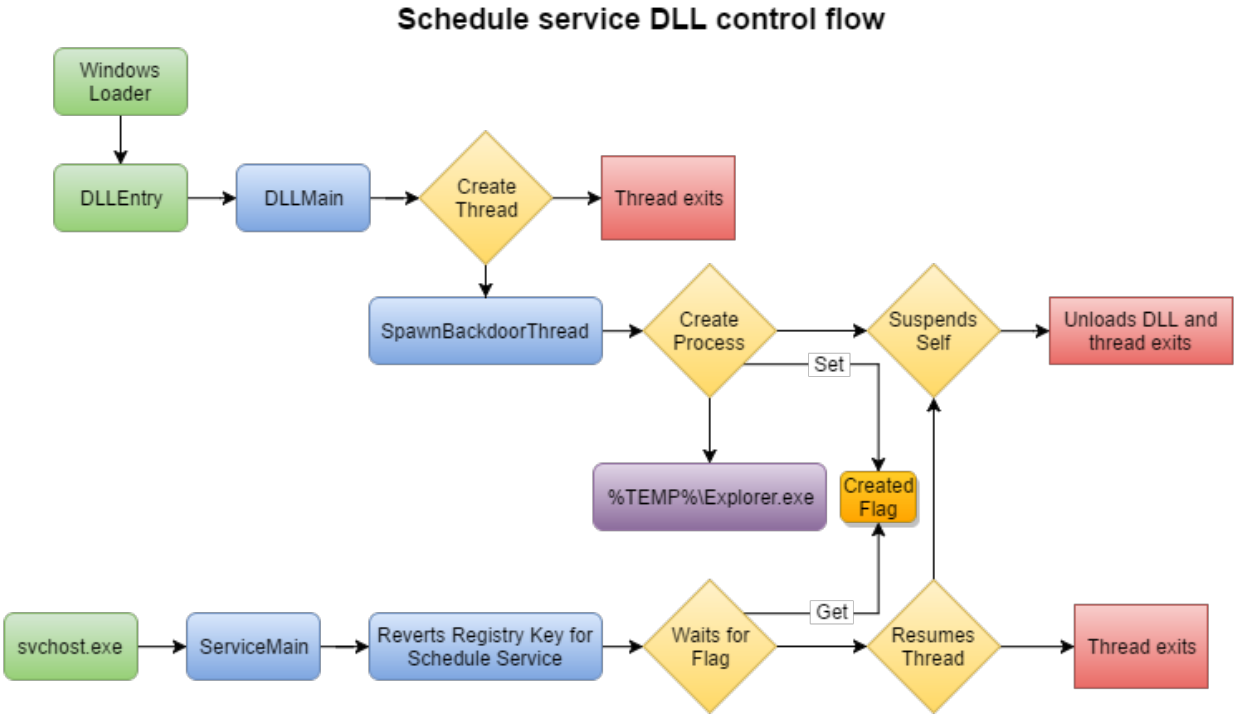

Figure 7. Flow of the malicious Schedule service

into a svchost.exe executable. The Schedule service is part of the NetworkService group, so the DLL will be loaded into the svchost.exe containing the other services for the group, and a new thread will be spawned to run the ServiceMain for that DLL. Additionally, as happens every time a DLL is loaded, the DLL's entry point (DLLMain, in this case) is called by the Windows loader in another thread.

The HDRoot authors chose to use the DLLMain function to start the backdoor process and ServiceMain to revert the service registry entry back to the original path. The DLLMain thread creates another thread running the function that is identified in the disassembly as SpawnBackdoorThread. That thread creates a process running the backdoor, which rkImage saved to %TEMP%\Explorer.exe. It then sets a global variable in the DLL to signal that it successfully launched the backdoor, and suspends itself before continuing.

Simultaneously the ServiceMain thread reverts the registry, and waits for the backdoor to start, sleeping and periodically checking for the flag to be set. After the flag is set it resumes the SpawnBackdoor thread, and then exits. In turn, the SpawnBackdoor thread unloads the DLL from memory and then exits itself.

This has the effect of both threads exiting and the DLL unloading at almost the exact same time, guaranteeing that the service manager will have to restart the service, causing the legitimate service DLL to be loaded and run from the patched registry entry. In all tests of this sample, not once did the real Windows service ever fail to start after running the bootkit. This is completely contrary to Kaspersky's claim that the bootkit breaks the service and that all the victims must just not have cared or noticed that the service failed to start.

8. Conclusions

The analysis of the HDRoot malware shows this sample to have a sophisticated and functional implementation. As is problematic and common with much work in the malware analysis field, Kaspersky's published findings were released without enough detail to verify. Additionally, no work was done to verify their findings prior to this analysis. The discrepancies between the conclusions in their report and the technical findings in this analysis leads the author to the conclusion than that they did their entire analysis with and presented research on a ten-year-old sample, but provided the signature indicators for the 2012 sample that had been in modern use. It also shows that the authors, who have been designated as the WINNTI group, have been around for a significant period of time, dating back to at least 2006 if the timestamps on Kaspersky's sample are to be believed.

The one stage of the attack in which the bootkit did not make good use of hiding techniques was in covering its tracks for the service and backdoor executables. Both the modified file hosting the DLL (in most tests C:\Windows\syswow64\C_932.NLS) and the backdoor in %TEMP%\Explorer.exe were left intact on the file system. However, it is likely that a sophisticated backdoor run by the bootkit would know to remove these two pieces of evidence after starting itself, and it may just have been a choice of segregation of duties made by the authors.

Another criticism that can be made is the extremely weak use of encryption. The XOR cipher is little more than obfuscation and was trivial to figure out even just looking at the encrypted sectors on the disk. It can be argued, however, that since the entire contents of the bootkit is code that will be decrypted before it can be run, there is little point in hiding it from anything but simple scans, as it could just be captured from memory by analysts. To that end, the simple cipher serves its purpose of not matching the signatures for executables or of a boot sector while on disk.

Overall the level of detail that went into making this malware was significant. It is capable of installing itself on any Windows version, 32 or 64-bit, dating back to Windows 2000, with the exception of newer installs using UEFI. The lack of UEFI support is unlikely to be an issue when targeting server systems, however, especially with virtualization on the rise, as very few virtual environments are virtualizing UEFI in their guests. The small touches, such as anticipating that the drive may have been repartitioned, show particular thought and foresight (or testing) by the authors. Clearly significant thought and work went into the creation of this bootkit, and it is a mistake to dismiss it as amateur. While different versions of the dropper are geared toward different targets (the observed sample here targets the Schedule service on 64-bit systems), the overall framework is very flexible. The choices made inside the dropper or when compiling the dropper are able to be tuned toward the target, choosing a service compatible with that version of Windows. This makes narrowing down the traits of the bootkit from which services it targets to be very difficult, as it is trivial for the authors to change their target.

Overall, this exercise shows that the analysis done by security companies would be more useful to the security field if their results were published in enough detail to be verifiable by outside sources. Most companies guard the particulars of their research as proprietary information, not sharing how their findings were discovered. But security defenders rely on these descriptions of attacks, not just file hash signatures, to

look for, detect, and build behavioral signatures for these newly discovered threats. It becomes ineffective for them to do so when they are provided misinformation and no resources to check these conclusions in a timely manner. While public disclosure of these threats is a great benefit to the security industry, on the whole it would be better served if more companies would publish in-depth and reproducible results.

9. References

[1] Securelist, "WINNTI: More than just a game," [Online]. Available: https://securelist.com/analysis/internal-threats-reports/37029/winnti-more-than-just-a-game/.

[2] E. A. II, "Backdoor Built With Aheadlib Used in Targeted Attacks?," Trend Micro, [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/backdoor-built-with-aheadlib-used-in-targeted-attacks/.

[3] malwr," [Online]. Available: https://malwr.com/analysis/NGFiNDBmMWNmYjM0NDVmZWIxNTg5OWFkMDUwYmIzNTQ/.

[4] Securelist, "I am HDRoot Part 1," [Online]. Available: https://securelist.com/analysis/publications/72275/i-am-hdroot-part-1/.

[5] X. Kovah, "Intermediate Intel x86," [Online]. Available: http://opensecuritytraining.info/IntermediateX86.html.

[6] AMD, "AMD64 Architecture Programmer's Manual, Volume 2," [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf.

[7] Microsoft, "National Language Support (NLS) API Reference," [Online]. Available: https://www.microsoft.com/resources/msdn/goglobal/default.mspx.

[8] Microsoft, "Kernel-Mode Device Driver TSP," [Online]. Available: https://msdn.microsoft.com/en-us/library/ms725209(v=vs.85).aspx.